Enrolment No./Seat No_____

GUJARAT TECHNOLOGICAL UNIVERSITY

BE- SEMESTER–III (NEW) EXAMINATION – WINTER 2024

Subject Code: 3130702 Date: 26-11-2024

Subject Name: Data Structures

Time: 10:30 AM TO 01:00 PM Total Marks: 70

Instructions:

1. **Attempt all questions.**
2. **Make suitable assumptions wherever necessary.**
3. **Figures to the right indicate full marks.**
4. **Simple and non-programmable scientific calculators are allowed.**

			Marks
Q.1	**(a)**	Differentiate Primitive and Non Primitive Data Structures	**03**
	(b)	Derive basic operation of stack and write C function to implement it.	**04**
	(c)	Explain Row Major and Column Major with Example.	**07**
Q.2	**(a)**	What is Sparse Matrix?	**03**
	(b)	Translate infix expression into its equivalent post fix expression: A*(B+D)/E-F*(G+H/K)	**04**
	(c)	Write a program to implement Circular queue and show how it differ from normal queue?	**07**
		OR	
	(c)	Explain the concept of Reverse Polish Notation (RPN) and describe the process for evaluating an RPN expression using a stack. Provide a detailed example of evaluating the following RPN expression: 5 1 2 + 4 * + 3 -	**07**
Q.3	**(a)**	Write an algorithm for Bubble sort.	**03**
	(b)	Sort 20,35,40,100,3,10,15 using insertion sort. Show all passes.	**04**
	(c)	Compare and contrast Depth First Search (DFS) and Breadth First Search (BFS) in terms of their algorithms, uses, and performance.	**07**
		OR	
Q.3	**(a)**	Write an algorithm for Merge Sort.	**03**
	(b)	Sort the given values using Quick Sort? 65, 70, 75, 80, 85, 60, 55, 50, 45. Show all passes.	**04**
	(c)	Describe the concept of hashing and explain different collision resolution techniques, including separate chaining and open addressing. Illustrate how each technique works with examples.	**07**
Q.4	**(a)**	Define Graph and list any 3 uses of graph.	**03**
	(b)	How to prove array is sequential and contiguous?	**04**
	(c)	Explain the concept of a graph and its representations. Compare adjacency matrix and adjacency list representations in terms of space complexity and use cases.	**07**
		OR	
Q.4	**(a)**	List any three uses of linked list.	**03**

	(b)	Write an algorithm to delete next element from doubly linked list from given position,	**04**
	(c)	Write an algorithm to implement singly linked list and its operation such as insert element at front, last and at any position.	**07**
Q.5	**(a)**	What is priority queue?	**03**
	(b)	Write an algorithm to implement queue using Linked List.	**04**
	(c)	Explain the concept of a binary search tree (BST) and its properties. Discuss how operations such as insertion, deletion, and searching are performed in a BST.	**07**

OR

Q.5	**(a)**	Define (1) Forest (2) Leaf Node (3) Tree	**03**
	(b)	Construct tree from Following • **In order Traversal:** D, B, E, A, F, C • **Pre order Traversal:** A, B, D, E, C, F	**04**
	(c)	Discuss the key concepts and techniques of AVL trees and 2-3 trees, focusing on their balance mechanisms and the impact on performance. Provide examples of insertions and deletions to illustrate how these trees maintain balance.	**07**

****

Q.1 (a) Differentiate Primitive and Non Primitive Data Structures (3 mark)

Introduction

A **data structure** is a way of organizing and storing data in a computer so that it can be accessed and modified efficiently. Based on the nature and complexity of data storage, data structures are broadly classified into **primitive** and **non-primitive** data structures. These two types differ in terms of data handling and usage.

Primitive Data Structures

Primitive data structures are **basic data types** provided directly by the programming language.

- They store **single values**.
- They occupy a **fixed amount of memory**.
- Operations on them are simple and fast.
- Examples include **int, float, char, double, and boolean**.

Non-Primitive Data Structures

Non-primitive data structures are **derived or composite structures** formed using primitive data types.

- They can store **multiple values**.
- They may occupy **variable memory space**.
- They are used to represent complex data.
- Examples include **arrays, structures, linked lists, stacks, queues, trees, and graphs**.

Difference Between Primitive and Non-Primitive Data Structures

- Primitive data structures store simple and single data items, whereas non-primitive data structures store complex and multiple data items.
- Primitive data structures have fixed memory size, whereas non-primitive data structures can have variable memory size.
- Primitive data structures are easy to process, whereas non-primitive data structures require complex operations.
- Primitive data structures are directly supported by programming languages, whereas non-primitive data structures are user-defined.

(b) Derive basic operation of stack and write C function to implement it. (4 mark)

Introduction

A **stack** is a **linear data structure** that follows the **Last In First Out (LIFO)** principle. In a stack, the last element inserted is the first one to be removed. Stacks are widely used in **expression evaluation, recursion, function calls, and undo mechanisms**.

Basic Operations of Stack

1. **Push (Insertion)**
    - Adds an element to the **top of the stack**.
    - Check for **stack overflow** (if the stack is full) before insertion.
2. **Pop (Deletion)**
    - Removes the **top element** from the stack.

- Check for **stack underflow** (if the stack is empty) before deletion.

3. **Peek / Top**
   - Returns the value of the **top element** without removing it.
   - Useful for inspection or evaluation.
4. **IsEmpty**
   - Checks if the stack is empty.
   - Returns true if the stack has no elements.
5. **IsFull**
   - Checks if the stack is full (in case of fixed-size stack).
   - Returns true if no more elements can be added.

C Program to Implement Stack Operations

```
#include <stdio.h>
#define MAX 50

int stack[MAX];
int top = -1;

/* Push Operation */
void push(int value) {
   if(top == MAX - 1) {
      printf("Stack Overflow\n");
   } else {
```

```
        top++;
        stack[top] = value;
        printf("%d pushed to stack\n", value);
    }
}

/* Pop Operation */
int pop() {
    if(top == -1) {
        printf("Stack Underflow\n");
        return -1;
    } else {
        int value = stack[top];
        top--;
        return value;
    }
}

/* Peek / Top Operation */
int peek() {
    if(top == -1) {
        printf("Stack is empty\n");
        return -1;
    } else {
```

```
        return stack[top];
    }
}

/* Main Function to Test Stack */
int main() {
    push(10);
    push(20);
    push(30);
    printf("Top element is %d\n", peek());
    printf("%d popped from stack\n", pop());
    printf("Top element is %d\n", peek());
    return 0;
}
```

Explanation of Program

- The stack is implemented using an **array** of fixed size MAX.
- The variable top keeps track of the **topmost element**.
- **Push** adds elements to the top, **pop** removes the top element, and **peek** returns the top element.
- **Overflow** and **underflow** are handled with checks.

(c) Explain Row Major and Column Major with Example. (7 mark)

Introduction / Definition:

In computer programming, arrays are data structures that store elements of the same type in a contiguous memory location. When

dealing with multi-dimensional arrays, such as two-dimensional arrays (matrices), the elements need to be stored in memory either row-wise or column-wise. These two methods are called **Row Major** and **Column Major** representations. Understanding these concepts is essential for efficient memory access, programming, and performance optimization in languages like C, C++, and Fortran.

---

1. Row Major Representation:

In **Row Major Order**, the elements of a multi-dimensional array are stored row by row in memory. That is, all elements of the first row are stored first, followed by the elements of the second row, and so on. This is the default method used by languages like C and C++.

Key Points:

- Memory allocation is sequential for rows.
- Efficient when row-wise operations are common.
- Element access formula for a 2D array A[m][n] in row major is:

Address(A[i][j])=Base Address+[(i×No. of Columns)+j]×Size of elem ent

Example:
Consider a 2×3 array:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

In **Row Major**, memory will store elements as:
1, 2, 3, 4, 5, 6

- First row [1 2 3] → stored first.
- Second row [4 5 6] → stored next.

This sequential storage allows fast access if we traverse row by row.

2. Column Major Representation:

In **Column Major Order**, the elements are stored column by column in memory. That is, all elements of the first column are stored first, followed by elements of the second column, and so on. Languages like Fortran and MATLAB use column-major order.

Key Points:

- Memory allocation is sequential for columns.
- Efficient for column-wise operations or calculations.
- Element access formula for a 2D array A[m][n] in column major is:

Address(A[i][j])=Base Address+[(j×No. of Rows)+i]×Size of element

Example:
For the same 2×3 array:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

In **Column Major**, memory will store elements as:
1, 4, 2, 5, 3, 6

- First column [1 4] → stored first.
- Second column [2 5] → stored next.
- Third column [3 6] → stored last
- This approach is suitable for algorithms that process data column-wise.
- ---
- **Comparison Between Row Major and Column Major:**

Feature	Row Major	Column Major
Storage Pattern	Row by row	Column by column
Common Languages	C, C++	Fortran, MATLAB

Feature	Row Major	Column Major
Memory Access Efficiency	Efficient for row traversal	Efficient for column traversal
Address Calculation	`Base + [(i × n) + j]`	`Base + [(j × m) + i]`

Applications:

- Row major is commonly used in **image processing** where row-wise scanning is frequent.
- Column major is useful in **matrix computations** in scientific computing and linear algebra, where column operations are more frequent.

Small Example Explained in Words:

Suppose we have a 2×2 matrix:

$$B = \begin{bmatrix} 7 & 8 \\ 9 & 10 \end{bmatrix}$$

- **Row Major Storage:** 7, 8, 9, 10
- **Column Major Storage:** 7, 9, 8, 10

Q.2 (a) What is Sparse Matrix?

Sparse Matrix

Definition / Introduction:
A **sparse matrix** is a type of matrix in which most of the elements are **zero**. In contrast, a matrix that has very few zero elements is called a **dense matrix**. Sparse matrices are common in scientific computing, engineering problems, and computer graphics, where large matrices often contain mostly zero values.

Key Points:

- Saves **memory space** because storing only non-zero elements reduces storage requirements.

- Improves **computational efficiency** as operations can be performed only on non-zero elements.
- Commonly represented using **special data structures** like:
  - **Triplet form** (row, column, value)
  - **Compressed Row Storage (CRS)**
  - **Compressed Column Storage (CCS)**

Purpose / Need:

- Large matrices with many zeros can waste memory if stored normally.
- Sparse matrix techniques help in **fast calculations** in linear algebra, graph algorithms, and scientific simulations.

Example:

$$\begin{bmatrix} 0 & 0 & 3 \\ 0 & 0 & 0 \\ 4 & 0 & 0 \end{bmatrix}$$

Here, only two elements (3 and 4) are non-zero; the rest are zeros, making it a sparse matrix.

Advantages:

- Reduces memory usage.
- Increases speed of matrix operations.
- Efficient for solving real-world large problems.

(b) Translate infix expression into its equivalent post fix expression: A*(B+D)/E-F*(G+H/K) (4 mark)

Introduction / Definition:
In computer science, an **infix expression** is the common arithmetic notation in which operators are written **between operands**. For

example, A + B. A **postfix expression** (also called Reverse Polish Notation) writes operators **after their operands**, for example, A B +.

Converting infix to postfix is important because computers and stack-based machines can **evaluate postfix expressions efficiently** without the need for parentheses, following operator precedence automatically. This is widely used in **compilers, calculators, and expression evaluation algorithms**.

Given Infix Expression:

$$A*(B+D)/E-F*(G+H/K)$$

Step-by-Step Conversion Using Stack Method:

Rules to remember:

1. If the symbol is an **operand** (A, B, C…), **add it directly** to the postfix expression.
2. If the symbol is **'('**, push it onto the stack.
3. If the symbol is **')'**, pop from the stack and add to postfix until '(' is encountered.
4. If the symbol is an **operator**, pop from the stack and add to postfix all operators with **higher or equal precedence**. Then push the current operator.

Step 1: Scan A → Operand → Add to postfix
Postfix: A

Step 2: Scan * → Operator → Push to stack
Stack: *

Step 3: Scan (→ Push to stack
Stack: * (

Step 4: Scan B → Operand → Add to postfix
Postfix: A B

Step 5: Scan + → Operator → Push to stack
Stack: * (+

Step 6: Scan D → Operand → Add to postfix
Postfix: A B D

Step 7: Scan) → Pop until (

- Pop + → Add to postfix → A B D +
- Pop (→ Discard
  Stack: *

Step 8: Scan / → Operator

- Precedence / = *, pop * → Add to postfix → A B D + *
- Push / → Stack: /

Step 9: Scan E → Operand → Add to postfix
Postfix: A B D + * E

Step 10: Scan - → Operator

- Precedence - < / → Pop / → Add to postfix → A B D + * E /
- Push - → Stack: -

Step 11: Scan F → Operand → Add to postfix
Postfix: A B D + * E / F

Step 12: Scan * → Operator → Push to stack
Stack: - *

Step 13: Scan (→ Push to stack
Stack: - * (

Step 14: Scan G → Operand → Add to postfix
Postfix: A B D + * E / F G

Step 15: Scan + → Operator → Push to stack
Stack: - * (+

Step 16: Scan H → Operand → Add to postfix
Postfix: A B D + * E / F G H

Step 17: Scan / → Operator → Higher precedence than + → Push
Stack: - * (+ /

Step 18: Scan K → Operand → Add to postfix
Postfix: A B D + * E / F G H K

Step 19: End of expression inside () → Pop till (

- Pop / → A B D + * E / F G H K /
- Pop + → A B D + * E / F G H K / +
- Pop (→ discard
  Stack: - *

Step 20: Pop remaining operators in stack

- Pop * → A B D + * E / F G H K / + *
- Pop - → A B D + * E / F G H K / + * -

Final Postfix Expression:

ABD+∗E/FGHK/+∗−

Key Points:

- Postfix expressions do not need parentheses.
- Evaluation is easier using a stack.
- Useful in compiler design and expression evaluation algorithms.
- Operator precedence and associativity are automatically handled using the stack method.

Small Example in Words:

If the infix is A + B * C, the postfix will be A B C * +.

- B * C is evaluated first due to higher precedence of *, then added to A.
- This demonstrates how postfix simplifies computation and stack evaluation.

(c) Write a program to implement Circular queue and show how it differ from normal queue? (7 mark)

Introduction / Definition:
A **queue** is a linear data structure that follows **FIFO (First In First Out)** order, where elements are inserted at the **rear** and removed from the **front**. A **normal (linear) queue** has a limitation: once the rear reaches the end of the array, no more elements can be inserted even if there are empty slots at the front due to deletions.

A **circular queue** overcomes this problem by treating the array as **circular**, meaning the position after the last index connects back to the first index. This allows efficient utilization of memory and avoids wastage of space in linear queues.

Key Features of Circular Queue:

- Follows **FIFO principle**.
- Front and rear pointers **wrap around** to the start when reaching the end.
- Efficient for **fixed-size memory buffers**.
- Prevents **overflow** when empty spaces are available at the beginning of the array.

Difference from Normal Queue:

Feature	Normal Queue	Circular Queue
Memory Usage	May waste space	Efficient, no wastage
Wrap Around	Not allowed	Allowed
Front and Rear Movement	Only forward	Forward with wrap-around
Overflow Condition	Rear reaches end	Rear reaches front

```
#include <stdio.h>
#define SIZE 5

int queue[SIZE];
int front = -1, rear = -1;

// Function to check if queue is full
int isFull() {
    return ((rear + 1) % SIZE == front);
}

// Function to check if queue is empty
int isEmpty() {
    return (front == -1);
```

```
}

// Function to insert element
void enqueue(int value) {
    if(isFull()) {
        printf("Queue Overflow\n");
        return;
    }
    if(front == -1) front = 0;
    rear = (rear + 1) % SIZE;
    queue[rear] = value;
    printf("%d inserted\n", value);
}

// Function to delete element
int dequeue() {
    if(isEmpty()) {
        printf("Queue Underflow\n");
        return -1;
    }
    int value = queue[front];
    if(front == rear) { // Queue has only one element
        front = rear = -1;
    } else {
```

```
        front = (front + 1) % SIZE;
    }
    return value;
}

// Function to display queue
void display() {
    if(isEmpty()) {
        printf("Queue is empty\n");
        return;
    }
    printf("Queue elements: ");
    int i = front;
    while(1) {
        printf("%d ", queue[i]);
        if(i == rear) break;
        i = (i + 1) % SIZE;
    }
    printf("\n");
}

int main() {
    enqueue(10);
    enqueue(20);
```

```
    enqueue(30);
    enqueue(40);
    enqueue(50); // Queue full
    display();
    printf("Deleted: %d\n", dequeue());
    printf("Deleted: %d\n", dequeue());
    enqueue(60);
    enqueue(70); // Wrap around insert
    display();
    return 0;
}
```

Explanation of Program:

- enqueue() inserts elements at the rear and wraps around using modulo.
- dequeue() removes elements from the front and updates the front pointer.
- isFull() and isEmpty() prevent overflow and underflow.
- display() shows all elements in proper order using circular traversal.

Applications of Circular Queue:

- **CPU scheduling** (round-robin algorithm)
- **Memory buffers** (keyboard buffer, printer queue)
- **Data streaming** (real-time data processing)

Example / Demonstration in Words:

Suppose we have a circular queue of size 5:

- Insert: 10, 20, 30, 40 → Queue = 10 20 30 40
- Delete: 10, 20 → Queue = 30 40
- Insert: 50, 60 → Queue wraps around → Queue = 30 40 50 60

In a normal queue, after inserting 40, further insertions would not be possible without shifting elements. Circular queue avoids this limitation.

OR

(c) Explain the concept of Reverse Polish Notation (RPN) and describe the process for evaluating an RPN expression using a stack. Provide a detailed example of evaluating the following RPN expression: 5 1 2 + 4 * + 3 – (7 mark)

Reverse Polish Notation (RPN) and Stack Evaluation

Introduction / Definition:
Reverse Polish Notation (RPN), also known as **postfix notation**, is a method of writing arithmetic expressions in which the **operators follow their operands**. Unlike infix notation, RPN does not require parentheses to define the order of operations because the sequence of operators and operands inherently preserves precedence.

RPN is widely used in **stack-based calculators**, **compiler design**, and **expression evaluation algorithms**, as it allows **efficient and unambiguous evaluation** using a stack data structure.

Concept of Stack-Based Evaluation:

A **stack** is a linear data structure that follows **LIFO (Last In First Out)** principle. Evaluating an RPN expression involves scanning the

expression **from left to right** and using a stack to store intermediate results.

Steps to Evaluate an RPN Expression:

1. Initialize an empty stack.
2. Scan the RPN expression **from left to right**.
3. For each symbol:
    - If it is an **operand**, push it onto the stack.
    - If it is an **operator** (+, -, *, /), pop the required number of operands from the stack, perform the operation, and push the result back onto the stack.
4. After the entire expression is scanned, the remaining element in the stack is the **final result**.

Given RPN Expression:

$$5\ 1\ 2\ +\ 4\ *\ +\ 3\ -$$

Step-by-Step Evaluation Using Stack:

Stack initially: empty

1. **Scan 5** → Operand → Push → Stack: [5]
2. **Scan 1** → Operand → Push → Stack: [5, 1]
3. **Scan 2** → Operand → Push → Stack: [5, 1, 2]
4. **Scan +** → Operator → Pop 2 and 1 → Calculate 1 + 2 = 3 → Push result → Stack: [5, 3]
5. **Scan 4** → Operand → Push → Stack: [5, 3, 4]
6. **Scan *** → Operator → Pop 4 and 3 → Calculate 3 * 4 = 12 → Push result → Stack: [5, 12]

7. **Scan +** → Operator → Pop 12 and 5 → Calculate 5 + 12 = 17 → Push result → Stack: [17]
8. **Scan 3** → Operand → Push → Stack: [17, 3]
9. **Scan -** → Operator → Pop 3 and 17 → Calculate 17 - 3 = 14 → Push result → Stack: [14]

Final Result: 14

Key Features of RPN Evaluation:

- **No parentheses needed**, operator precedence is inherently handled.
- Uses **stack efficiently** for intermediate results.
- Each operator applies to the **two most recent operands**.
- Simple algorithm suitable for **computer and calculator implementation**.

Advantages:

- Reduces complexity in parsing expressions.
- Efficient memory usage using stack.
- Easy to implement in **hardware and software calculators**.

Disadvantages:

- Not intuitive for humans compared to infix notation.
- Requires **stack-based computation** to evaluate.

Example in Words:

The expression 5 1 2 + 4 * + 3 - is equivalent to the infix expression:

- First, 1 + 2 = 3

- Then 3 * 4 = 12
- Add 5 + 12 = 17
- Subtract 17 - 3 = 14
  This step-by-step process matches the **stack evaluation method**.

Q.3 (a) Write an algorithm for Bubble sort. (3 mark)

Bubble Sort Algorithm

Introduction / Definition:
Bubble Sort is a simple **comparison-based sorting algorithm** used to arrange elements of an array in **ascending or descending order**. In this algorithm, adjacent elements are compared and swapped if they are in the wrong order. The process repeats until the array is fully sorted. Bubble Sort is easy to understand and implement, but it is not very efficient for large datasets.

Algorithm Steps:

1. **Start**
2. Initialize an array of n elements.
3. Repeat for i = 0 to n-1:
    - For j = 0 to n-i-2:
        - Compare arr[j] and arr[j+1].
        - If arr[j] > arr[j+1], **swap** them.
4. Continue the process until no more swaps are needed.
5. **End**

Key Points / Advantages:

- Simple and easy to implement.
- **Stable sort** (does not change relative order of equal elements).
- Best for **small datasets** or **almost sorted arrays**.
- Time Complexity:
  - Worst Case: **O(n²)**
  - Best Case: **O(n)** (if already sorted)

Example in Words:

Given array: [5, 3, 8, 2]

- Pass 1: Compare 5 & 3 → swap → [3, 5, 8, 2]
  Compare 5 & 8 → no swap
  Compare 8 & 2 → swap → [3, 5, 2, 8]
- Pass 2: Compare 3 & 5 → no swap
  Compare 5 & 2 → swap → [3, 2, 5, 8]
- Pass 3: Compare 3 & 2 → swap → [2, 3, 5, 8] → Sorted

(b) Sort 20,35,40,100,3,10,15 using insertion sort. Show all passes. (4 mark)

Insertion Sort

Introduction / Definition:
Insertion Sort is a simple sorting algorithm that builds the final sorted array **one element at a time**. It picks an element from the unsorted portion and places it at the correct position in the sorted portion. It is efficient for **small datasets** and partially sorted arrays.

Given Array:

[20,35,40,100,3,10,15]

Step-by-Step Passes:

Initial Array: [20, 35, 40, 100, 3, 10, 15]

Pass 1: Consider 35 → Compare with 20 → 35 > 20 → Place 35 → [20, 35, 40, 100, 3, 10, 15]

Pass 2: Consider 40 → Compare with 35 → 40 > 35 → Place 40 → [20, 35, 40, 100, 3, 10, 15]

Pass 3: Consider 100 → Compare with 40 → 100 > 40 → Place 100 → [20, 35, 40, 100, 3, 10, 15]

Pass 4: Consider 3 → Compare with 100 → 3 < 100 → shift 100
Compare with 40 → 3 < 40 → shift 40
Compare with 35 → 3 < 35 → shift 35
Compare with 20 → 3 < 20 → shift 20 → Place 3
Array: [3, 20, 35, 40, 100, 10, 15]

Pass 5: Consider 10 → Compare with 100 → shift 100
Compare with 40 → shift 40
Compare with 35 → shift 35
Compare with 20 → shift 20
Compare with 3 → 10 > 3 → Place 10
Array: [3, 10, 20, 35, 40, 100, 15]

Pass 6: Consider 15 → Compare with 100 → shift 100
Compare with 40 → shift 40
Compare with 35 → shift 35
Compare with 20 → shift 20
Compare with 10 → 15 > 10 → Place 15
Array: [3, 10, 15, 20, 35, 40, 100]

Sorted Array:

[3,10,15,20,35,40,100]

Key Points / Features:

- Builds sorted array **one element at a time**.
- **Stable sort** (does not change relative order of equal elements).
- Best for **small datasets** or **nearly sorted arrays**.
- Time Complexity:
  - Worst Case: **$O(n^2)$**
  - Best Case: **O(n)**

(c) Compare and contrast Depth First Search (DFS) and Breadth First Search (BFS) in terms of their algorithms, uses, and performance.(7 mark)

Introduction / Definition:
Depth First Search (DFS) and **Breadth First Search (BFS)** are two fundamental graph traversal algorithms used to explore nodes and edges in a graph.

- **DFS** explores a graph **by going as deep as possible along a branch before backtracking**.
- **BFS** explores a graph **level by level**, visiting all neighbors of a node before moving to the next level.

These algorithms are essential in **graph theory, network analysis, AI, and pathfinding applications**.

Depth First Search (DFS):

Algorithm / Steps:

1. Start from a selected node.
2. Mark the node as visited and push it onto a **stack** (or use recursion).

3. Explore an unvisited adjacent node, mark it, and continue recursively.
4. If no unvisited adjacent nodes remain, **backtrack** to the previous node.
5. Repeat until all reachable nodes are visited.

Key Features:

- Uses **stack** (or recursion) for backtracking.
- Traverses **deep into the graph** before exploring siblings.
- Can find **connected components**, paths, and cycles.

Applications:

- Solving **mazes and puzzles**.
- **Topological sorting**.
- Detecting **cycles** in a graph.

Performance:

- Time Complexity: **O(V + E)** (V = vertices, E = edges)
- Space Complexity: O(V) (for stack or recursion)

Breadth First Search (BFS):

Algorithm / Steps:

1. Start from a selected node.
2. Mark the node as visited and enqueue it into a **queue**.
3. Dequeue a node, visit all its unvisited adjacent nodes, mark them visited, and enqueue them.
4. Repeat until the queue is empty.

Key Features:

- Uses a **queue** for level-wise traversal.
- Explores **all neighbors at the current depth** before going deeper.
- Finds the **shortest path** in unweighted graphs.

Applications:

- **Shortest path finding** in unweighted graphs (e.g., maps, networks).
- **Peer-to-peer networking**.
- **Web crawling** and social network analysis.

Performance:

- Time Complexity: **O(V + E)**
- Space Complexity: O(V) (for queue)

Comparison Table:

Feature	**DFS**	**BFS**
Traversal Method	Deep into branches first	Level by level
Data Structure Used	Stack (or recursion)	Queue
Path Finding	Not guaranteed shortest path	Guarantees shortest path in unweighted graphs
Memory Requirement	Less for sparse graphs, can grow with recursion	Can be high for wide graphs

Feature	DFS	BFS
Use Case	Cycle detection, topological sort, maze solving	Shortest path, social networks, broadcast routing
Backtracking	Yes	No

Example in Words:

Consider a graph:

```
  A
 / \
B   C
|   |
D   E
```

- **DFS starting at A:** A → B → D → C → E
  (Goes deep into B-D branch first before visiting C-E)
- **BFS starting at A:** A → B → C → D → E
  (Visits all nodes level by level from the starting node)

OR

Q.3 (a) Write an algorithm for Merge Sort. (3 mark)

Merge Sort Algorithm

Introduction / Definition:
Merge Sort is a **divide-and-conquer** sorting algorithm that divides an array into two halves, sorts them recursively, and then **merges the sorted halves**. It is more efficient than simple sorting algorithms like Bubble or Insertion sort, especially for **large datasets**, and guarantees **O(n log n) time complexity** in all cases.

Algorithm Steps:

1. **Start**
2. If the array has **one or zero elements**, it is already sorted → return.
3. Divide the array into two halves: **left** and **right**.
4. Recursively apply **Merge Sort** on the left half.
5. Recursively apply **Merge Sort** on the right half.
6. **Merge** the two sorted halves into a single sorted array:
    - Compare the first elements of both halves.
    - Place the smaller element into the merged array.
    - Repeat until all elements are merged.
7. Return the **sorted array**.
8. **End**

Key Points / Advantages:

- **Stable sort**: preserves order of equal elements.
- Time Complexity:
    - Best, Worst, Average: **O(n log n)**
- Space Complexity: **O(n)** (requires temporary arrays for merging).
- Efficient for **large datasets**.
- Example: [38, 27, 43, 3] → Divide → [38, 27], [43, 3] → Merge → [27, 38, 3, 43] → Final sorted [3, 27, 38, 43].

(b) Sort the given values using Quick Sort? 65, 70, 75, 80, 85, 60, 55, 50, 45. Show all passes.(4 mark)

Quick Sort

Introduction / Definition:
Quick Sort is a **divide-and-conquer** sorting algorithm. It selects a **pivot element** from the array and partitions the array such that all elements **less than the pivot** go to the left, and all elements **greater than the pivot** go to the right. The same process is applied recursively to the sub-arrays. Quick Sort is efficient for **large datasets** with average time complexity **O(n log n)**.

Given Array:

$$[65, 70, 75, 80, 85, 60, 55, 50, 45]$$

Step-by-Step Passes (Using Last Element as Pivot):

Pass 1: Pivot = 45

- Partition → elements < 45: none, elements > 45: [65, 70, 75, 80, 85, 60, 55, 50]
- Place pivot → [45, 70, 75, 80, 85, 60, 55, 50, 65]
- Left sub-array: empty
- Right sub-array: [70, 75, 80, 85, 60, 55, 50, 65]

Pass 2: Pivot = 65 (last of right sub-array)

- Partition → elements < 65: [60, 55, 50], elements > 65: [70, 75, 80, 85]
- Place pivot → [45, 60, 55, 50, 65, 70, 75, 80, 85]

Pass 3: Left sub-array of 65: [60, 55, 50], Pivot = 50

- Partition → elements < 50: none, elements > 50: [60, 55]
- Place pivot → [45, 50, 55, 60, 65, 70, 75, 80, 85]

Pass 4: Right sub-array of 50: [60, 55], Pivot = 55

- Partition → elements < 55: none, elements > 55: [60]
- Place pivot → [45, 50, 55, 60, 65, 70, 75, 80, 85]

Pass 5: Right sub-array of 65: [70, 75, 80, 85], Pivot = 85

- Partition → elements < 85: [70, 75, 80], elements > 85: none
- Place pivot → [45, 50, 55, 60, 65, 70, 75, 80, 85]

Pass 6: Sub-array [70, 75, 80], Pivot = 80

- Partition → elements < 80: [70, 75], elements > 80: none
- Place pivot → [45, 50, 55, 60, 65, 70, 75, 80, 85]

Pass 7: Sub-array [70, 75], Pivot = 75

- Partition → elements < 75: [70], elements > 75: none
- Place pivot → [45, 50, 55, 60, 65, 70, 75, 80, 85]

Sorted Array:

[45,50,55,60,65,70,75,80,85]

Key Points / Features:

- **Divide-and-conquer approach** using pivot.
- Average Time Complexity: **O(n log n)**
- Worst Case (already sorted with bad pivot): **O(n²)**
- In-place sorting → requires minimal extra memory.
- Efficient for **large datasets**.

(c) Describe the concept of hashing and explain different collision resolution techniques, including separate chaining and open addressing. Illustrate how each technique works with examples. (7 mark)

Introduction / Definition:
Hashing is a technique used to store and retrieve data efficiently in a **hash table**. In hashing, a **hash function** converts a key into an index (or address) in the hash table. This allows **constant time (O(1))** access for insertion, deletion, and search in ideal scenarios. Hashing is widely used in **databases, password storage, caching, and compiler symbol tables**.

Hash Function:

A **hash function** takes a key as input and returns an integer index in the hash table.
Example:

$$h(k) = k \bmod m$$

Where k is the key and m is the size of the hash table.

Collision in Hashing:

A **collision** occurs when **two keys produce the same index** in the hash table. Efficient collision resolution techniques are required to maintain fast access.

---

1. Separate Chaining:

Concept:

- Each slot of the hash table contains a **linked list**.
- If multiple keys hash to the same index, they are stored in the linked list at that slot.

Steps / Example:

- Hash table size = 5, keys = [12, 15, 25, 35]

- Hash function: h(k) = k mod 5

Key	Hash Index	Action
12	2	Insert 12 at index 2
15	0	Insert 15 at index 0
25	0	Collision → add 25 to linked list at index 0 → [15 → 25]
35	0	Collision → add 35 to linked list at index 0 → [15 → 25 → 35]

Advantages:

- Simple to implement.
- Works well when table is **sparsely filled**.

Disadvantages:

- Extra memory for linked lists.
- Traversal of long chains can slow down operations.

---

2. Open Addressing:

Concept:

- All keys are stored **within the hash table itself**.
- On collision, the algorithm **probes for the next empty slot** using a probing sequence.

Types of Open Addressing:

1. **Linear Probing:**

- If slot i is occupied, try (i+1) mod m, (i+2) mod m, … until an empty slot is found.
- Example: Hash table size = 5, keys = [12, 15, 25], h(k) = k mod 5
  - 12 → index 2 → insert
  - 15 → index 0 → insert
  - 25 → index 0 → collision → linear probe → index 1 empty → insert 25

2. **Quadratic Probing:**
   - If collision occurs, try slots i + 1², i + 2², i + 3², ... mod m
   - Reduces **clustering** seen in linear probing.
3. **Double Hashing:**
   - Use a second hash function to calculate **step size**:

$$i = (h_1(k) + j \cdot h$$

1.

   -
   - More uniform distribution and reduces clustering.

Advantages of Open Addressing:

- No extra memory for linked lists.
- All elements stored in one array.

Disadvantages:

- Clustering may occur (linear probing).
- Performance degrades at high load factors (>70%).

Comparison of Techniques:

Feature	Separate Chaining	Open Addressing
Storage	Uses linked lists	Uses hash table only
Memory Overhead	Extra memory for chains	No extra memory
Performance at High Load	Slower due to long chains	Slower due to probing
Simplicity	Simple to implement	Slightly more complex

Summary / Key Points:

- **Hashing** allows fast insertion, deletion, and search using **hash functions**.
- **Collision resolution** is essential to handle cases when multiple keys hash to the same index.
- **Separate Chaining** stores collisions in linked lists.
- **Open Addressing** stores all elements in the table using probing techniques.
- Choice of technique depends on **memory availability and expected load factor**.

Q.4 (a) Define Graph and list any 3 uses of graph.(3 mark)

Graph

Definition / Introduction:
A **graph** is a non-linear data structure consisting of a set of **vertices (nodes)** and a set of **edges** that connect pairs of vertices. Graphs are

used to represent relationships between objects and can be **directed** (edges have direction) or **undirected** (edges have no direction). They are widely used in **computer science, networking, and real-world modeling**.

Key Points:

- **Vertices (V):** Represent objects or entities.
- **Edges (E):** Represent relationships or connections between vertices.
- **Graph Types:**
  - **Directed Graph (Digraph):** Edges have direction.
  - **Undirected Graph:** Edges have no direction.

Uses of Graph:

1. **Social Networks:** Representing users as vertices and friendships as edges.
2. **Computer Networks / Internet:** Nodes as computers/routers, connections as edges.
3. **Shortest Path / Navigation Systems:** Cities as vertices and roads as edges to find shortest route.

(b) How to prove array is sequential and contiguous?(4 mark)

Introduction / Definition:
An **array** is a collection of elements of the same type stored in **contiguous memory locations**. This arrangement allows **direct access** to any element using its **index**. Proving that an array is

sequential and contiguous ensures that the elements are stored in **consecutive memory addresses**.

Steps to Prove Array is Sequential and Contiguous:

1. **Understand Base Address:**
    - Let base be the memory address of the first element, arr[0].
2. **Access Formula:**
    - For an array of type T with element size sizeof(T), the address of element arr[i] is:

1.

    -
    - This formula assumes **sequential storage**.
2. **Check Differences Between Addresses:**
    - Compute the difference between consecutive elements:

$$\text{Address}(arr[i+1]) - \text{Address}(arr[i]) = \text{sizeof(T)}$$

    - If the difference is consistent for all elements, the array is **contiguous**.
3. **Example:**
    - Suppose arr[5] of type int (size 4 bytes) is stored starting at address 1000.
    - Expected addresses:
        - arr[0] = 1000
        - arr[1] = 1004

        - arr[2] = 1008
        - arr[3] = 1012
        - arr[4] = 1016
    - The consistent difference 4 bytes proves **sequential and contiguous storage**.

Key Points:

- Sequential addresses allow **direct index-based access**.
- Contiguous memory ensures **efficient memory usage** and **faster access**.
- This property is crucial for **pointer arithmetic** in C/C++ and similar languages.

(c) Explain the concept of a graph and its representations. Compare adjacency matrix and adjacency list representations in terms of space complexity and use cases. (7 mark)

Introduction / Definition:
A **graph** is a non-linear data structure consisting of a set of **vertices (nodes)** and a set of **edges** connecting pairs of vertices. Graphs are used to model real-world problems such as **social networks, transportation networks, and computer networks**. They can be **directed** (edges have direction) or **undirected** (edges have no direction).

Graphs can be represented in memory in different ways, primarily using **adjacency matrices** or **adjacency lists**, depending on efficiency and application needs.

---

1. Adjacency Matrix Representation

Concept:

- Uses a **2D array** of size V × V (V = number of vertices).
- Element matrix[i][j] is 1 (or weight) if there is an edge from vertex i to vertex j, otherwise 0.

Example:
Graph: Vertices {A, B, C}, Edges {A-B, B-C}

	A	**B**	**C**
A	0	1	0
B	0	0	1
C	0	0	0

Advantages:

- Easy to check **if an edge exists** between two vertices (O(1)).
- Simple and straightforward to implement.

Disadvantages:

- Requires **O(V²) space**, even if the graph is sparse.
- Inefficient for **large sparse graphs**.

---

2. Adjacency List Representation

Concept:

- Uses an **array of lists**. Each vertex has a list containing all vertices adjacent to it.
- More space-efficient for sparse graphs.

Example:
Graph: Vertices {A, B, C}, Edges {A-B, B-C}

- A → B
- B → C
- C → -

Advantages:

- Space-efficient: **O(V + E)**, where E = number of edges.
- Efficient for **traversal algorithms** like DFS and BFS.

Disadvantages:

- Checking the presence of a specific edge may take **O(V)** in worst case.
- Slightly more complex to implement than a matrix.

Comparison Table:

Feature	**Adjacency Matrix**	**Adjacency List**
Space Complexity	$O(V^2)$	O(V + E)
Edge Lookup	O(1)	O(k) (k = number of neighbors)
Memory Usage	High for sparse graphs	Low for sparse graphs
Best Use Case	Dense graphs	Sparse graphs
Implementation	Simple	Slightly complex

OR

Q.4 (a) List any three uses of linked list. (3 mark)

Uses of Linked List

Introduction / Definition:
A **linked list** is a linear data structure in which elements, called **nodes**, are connected using **pointers**. Unlike arrays, linked lists do not require contiguous memory locations. Each node contains **data** and a **pointer** to the next node (or previous node in doubly linked lists). Linked lists are widely used in scenarios where **dynamic memory allocation** and **flexible data insertion/deletion** are required.

Three Uses of Linked List:

1. **Dynamic Memory Allocation:**
    - Useful when the **size of data is not known in advance**.
    - Nodes can be allocated or deallocated **at runtime** without shifting elements.
2. **Implementation of Data Structures:**
    - Linked lists are used to implement **stacks, queues, graphs, and hash tables**.
    - For example, a **stack** can be implemented using a **singly linked list**, allowing push and pop operations efficiently.
3. **Efficient Insertion and Deletion:**
    - Inserting or deleting nodes at **any position** is faster compared to arrays (no need to shift elements).
    - Useful in applications like **music playlists, undo operations in editors, and operating system task scheduling**.

(b) Write an algorithm to delete next element from doubly linked list from given position,(4 mark)

Introduction / Definition:
A **doubly linked list (DLL)** is a data structure where each node contains **data, a pointer to the next node, and a pointer to the previous node**. Deleting a node involves updating the pointers of the **previous and next nodes** to maintain the list structure.

Algorithm:

Input: Pointer to head of the DLL and the given position pos
Output: DLL after deletion of the node **next to the given position**

1. **Start**
2. Check if the list is **empty** (head == NULL) → If yes, print "List is empty" and **exit**.
3. Initialize a pointer temp = head.
4. Traverse the list to reach the node at **position pos**:
5. for i = 1 to pos:
6. if temp == NULL:
7. print "Position not found"
8. exit
9. temp = temp->next
10. Check if the **next node exists** (temp->next != NULL) → If not, print "No next node to delete" and exit.
11. Let delNode = temp->next.
12. Update pointers to remove delNode from the list:
    - temp->next = delNode->next

- If delNode->next != NULL, then delNode->next->prev = temp

13. Free memory of delNode.
14. **End**

Key Points:

- Ensure **boundary checks** to avoid null pointer errors.
- Efficient for **dynamic insertion/deletion** compared to arrays.
- Time complexity: **O(n)** (traverse to position pos)
- Memory: **O(1)** (deletion uses constant extra memory)

Example in Words:

- Given DLL: 10 <-> 20 <-> 30 <-> 40 <-> 50
- Position pos = 2 → Node with data 20
- Next node to delete: 30
- After deletion: 10 <-> 20 <-> 40 <-> 50

(c) Write an algorithm to implement singly linked list and its operation such as insert element at front, last and at any position. (7 mark)

Introduction / Definition:
A **singly linked list (SLL)** is a linear data structure in which each node contains **data** and a **pointer to the next node**. The last node points to NULL, indicating the end of the list. Unlike arrays, SLL does **not require contiguous memory**, and allows **dynamic memory allocation**, efficient insertion, and deletion at any position.

Node Structure:

```
struct Node {
    int data;
    struct Node* next;
};
```

Head Pointer:

- Points to the **first node** of the list.

Algorithm for Singly Linked List Operations:

1. Insertion at Front

Steps:

1. Create a new node newNode.
2. Assign newNode->data = value.
3. Set newNode->next = head.
4. Update head = newNode.
5. End.

---

2. Insertion at Last (End)

Steps:

1. Create a new node newNode.
2. Assign newNode->data = value and newNode->next = NULL.
3. If head == NULL, set head = newNode.
4. Else, traverse the list to the **last node** (temp->next == NULL).
5. Set temp->next = newNode.
6. End.

---

3. Insertion at Any Position

Steps:

1. Create a new node newNode.
2. Assign newNode->data = value.
3. Input the **position pos** where insertion is required.
4. If pos == 1, perform **insertion at front**.
5. Else, traverse the list to reach the node at position pos-1 (temp).
6. Set newNode->next = temp->next.
7. Set temp->next = newNode.
8. End.

Key Points / Features:

- Dynamic size: No fixed size limitation.
- Memory efficient: Only allocate memory when required.
- Supports insertion at **front, end, or any position**.
- Time complexity:
    - Insertion at front: O(1)
    - Insertion at end or any position: O(n)

Example in Words:

- **Initially empty list**: NULL
- **Insert 10 at front** → List: 10
- **Insert 20 at end** → List: 10 -> 20

- **Insert 15 at position 2** → List: 10 -> 15 -> 20

Q.5 (a) What is priority queue? (3 mark)

Introduction / Definition:
A **priority queue** is a special type of **queue data structure** in which each element is associated with a **priority**. Unlike a regular queue where elements are processed in **FIFO (First In First Out)** order, in a priority queue, elements with **higher priority** are dequeued before elements with **lower priority**, regardless of their arrival time.

Key Points / Features:

- Each element has a **priority value** along with its data.
- Two types of priority queues:
    1. **Max-priority queue:** Higher priority elements are served first.
    2. **Min-priority queue:** Lower priority elements are served first.
- Can be implemented using **arrays, linked lists, or heaps**.
- Useful in **scheduling, operating systems, and simulation systems**.

Example in Words:

Suppose a queue of tasks with priorities:

Task	Priority
T1	2

Task Priority

T2 5

T3 1

- **Dequeue order (max-priority):** T2 → T1 → T3

(b) Write an algorithm to implement queue using Linked List. (4 mark)

Queue Implementation Using Linked List

Introduction / Definition:
A **queue** is a linear data structure that follows **FIFO (First In First Out)** order. Using a **linked list** to implement a queue allows **dynamic memory allocation**, avoiding the fixed size limitation of arrays. A queue has two main operations:

1. **Enqueue:** Insert element at the rear.
2. **Dequeue:** Remove element from the front.

Node Structure:

```
struct Node {
   int data;
   struct Node* next;
};
```

Pointers Used:

- front: Points to the first element of the queue.
- rear: Points to the last element of the queue.

Algorithm:

1. Enqueue (Insert at Rear)

1. Create a new node newNode and assign newNode->data = value.
2. Set newNode->next = NULL.
3. If front == NULL and rear == NULL, set both front = rear = newNode.
4. Else, set rear->next = newNode and rear = newNode.
5. End.

2. Dequeue (Delete from Front)

1. If front == NULL, print "Queue Underflow" and exit.
2. Store temp = front.
3. Set front = front->next.
4. If front == NULL, set rear = NULL.
5. Free memory of temp.
6. End.

3. Display Queue (Optional)

1. Start from front.
2. Traverse each node and print data until NULL is reached.
3. End.

Key Points / Features:

- Dynamic size: No fixed limit like array-based queue.
- **Time Complexity:**
    - Enqueue: O(1)

- Dequeue: O(1)

- Useful for **CPU scheduling, printer queue, and simulation systems**.

(c) Explain the concept of a binary search tree (BST) and its properties. Discuss how operations such as insertion, deletion, and searching are performed in a BST. (7 mark)

Binary Search Tree (BST) and Its Operations

Introduction / Definition:
A **Binary Search Tree (BST)** is a type of **binary tree** in which each node contains a key, and the following properties are maintained:

1. The **left subtree** of a node contains keys **less than** the node's key.
2. The **right subtree** of a node contains keys **greater than** the node's key.
3. Both left and right subtrees are themselves BSTs.

BSTs are widely used for **efficient searching, sorting, and dynamic data storage**.

Properties of a BST:

- **Node arrangement:** Left < Root < Right.
- **No duplicate keys** (in standard BST).
- **In-order traversal** of BST gives keys in **ascending order**.
- Average height = **O(log n)** for balanced BST, worst-case height = **O(n)**.

Operations on BST:

1. Insertion:

Steps:

1. Start at the root.
2. Compare the new key with the current node's key.
    - If new key < current key → move to left child.
    - If new key > current key → move to right child.
3. Repeat until a **NULL position** is found.
4. Insert the new node at the NULL position.

Example: Insert 15 into BST with root 20:

- 15 < 20 → go left
- Left child is 10 → 15 > 10 → go right → insert 15

---

2. Deletion:

Steps:

- Locate the node to delete. There are three cases:

1. **Leaf node:** Simply remove it.
2. **Node with one child:** Replace the node with its child.
3. **Node with two children:**
    - Find the **in-order successor** (smallest node in right subtree) or **in-order predecessor** (largest node in left subtree).
    - Replace node's key with successor/predecessor key.
    - Delete the successor/predecessor node.

---

3. Searching:

Steps:

1. Start at the root.
2. Compare the key to be searched with current node's key.
    - If equal → key found.
    - If less → go left.
    - If greater → go right.
3. Repeat until key is found or NULL is reached.

Time Complexity:

- Average case: O(log n)
- Worst case (skewed tree): O(n)

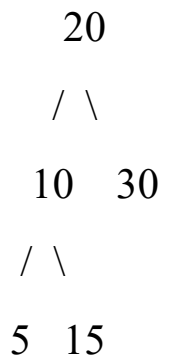
Key Points / Advantages:

- Efficient **search, insertion, and deletion** in balanced BST.
- In-order traversal gives **sorted elements**.
- Used in **database indexing, symbol tables, and searching algorithms**.

Example in Words:

Given keys: 20, 10, 30, 5, 15

- BST structure:



- Search 15 → Compare 15 < 20 → go left → Compare 15 > 10 → go right → Found.
- Insert 25 → 25 < 20? No → go right → 25 < 30 → go left → insert 25.
- Delete 10 → Node has two children → in-order successor = 15 → Replace 10 with 15 → Delete original 15.

OR

Q.5 (a) Define (1) Forest (2) Leaf Node (3) Tree (3 mark)

1. Tree:
A **tree** is a non-linear hierarchical data structure consisting of **nodes** connected by **edges**, with the following properties:

- There is **one node called the root** from which all nodes descend.
- Each node may have **zero or more child nodes**.
- Trees are used to represent hierarchical relationships, such as **file systems, organizational charts, and expression trees**.

---

2. Leaf Node:
A **leaf node** is a node in a tree that **does not have any children**. It is also called a **terminal node**.

- Leaf nodes are located at the **bottom of the tree**.
- Example: In a tree of employees, employees without subordinates are leaf nodes.

---

3. Forest:
A **forest** is a **collection of one or more disjoint trees**.

- If multiple trees are not connected by edges, they together form a forest.

- Converting a forest into a single tree can be done by adding a **new root node** and connecting it to the roots of all trees in the forest.

Key Points:

- A tree is a **single hierarchical structure**.
- Leaf nodes are **terminal points** of the tree.
- A forest is a **set of multiple trees**.

(b) Construct tree from Following

• In order Traversal: D, B, E, A, F, C

• Pre order Traversal: A, B, D, E, C, F (4 mark)

Introduction / Definition:
A **binary tree** can be uniquely constructed if **preorder** (or postorder) and **inorder** traversals are known.

- **Preorder Traversal:** Root → Left → Right
- **Inorder Traversal:** Left → Root → Right

Given:

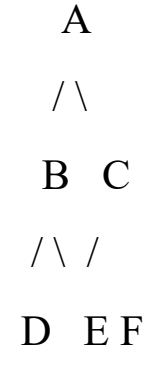
- **Inorder:** D, B, E, A, F, C
- **Preorder:** A, B, D, E, C, F

Steps to Construct the Tree:

1. **Start with Preorder:**
    - The first element in preorder is the **root**.
    - Root = A

2. **Locate Root in Inorder:**
    - Inorder: D, B, E, **A**, F, C
    - Left subtree (elements before A) = D, B, E
    - Right subtree (elements after A) = F, C
3. **Construct Left Subtree:**
    - Preorder elements for left subtree = B, D, E
    - Root of left subtree = B
    - Locate B in inorder (D, B, E):
        - Left child = D
        - Right child = E
4. **Construct Right Subtree:**
    - Preorder elements for right subtree = C, F
    - Root of right subtree = C
    - Locate C in inorder (F, C):
        - Left child = F
        - Right child = NULL

Constructed Tree Structure (in words):



- **Root:** A

- **Left Subtree:** B → D (left), E (right)
- **Right Subtree:** C → F (left), NULL (right)

Key Points:

- Preorder gives **root node first**, guiding construction.
- Inorder divides the tree into **left and right subtrees**.
- Recursive construction continues for each subtree.

(c) Discuss the key concepts and techniques of AVL trees and 2-3 trees, focusing on their balance mechanisms and the impact on performance. Provide examples of insertions and deletions to illustrate how these trees maintain balance.(07 mark)

AVL Trees and 2-3 Trees

Introduction / Definition:
Balanced search trees are designed to maintain **height balance** to ensure efficient **search, insertion, and deletion** operations. Two common types are **AVL trees** and **2-3 trees**.

- **AVL Tree:** A **binary search tree** where the difference between heights of left and right subtrees of every node is at most **1**.
- **2-3 Tree:** A **multi-way search tree** where every node can have **2 or 3 children** and **1 or 2 keys**, ensuring all leaves are at the same depth.

Key Concepts:

1. AVL Tree:

- Maintains **balance factor** at each node:

$$\text{Balance Factor} = \text{Height(left subtree)} - \text{Height(right subtree)}$$

- **Balance Factor** must be **-1, 0, or +1**.
- **Rotations** are performed to restore balance after insertion or deletion:
    1. **Single Right Rotation (LL Rotation)**
    2. **Single Left Rotation (RR Rotation)**
    3. **Left-Right Rotation (LR Rotation)**
    4. **Right-Left Rotation (RL Rotation)**

Example – Insertion:

- Insert 10, 20, 30 in an empty AVL tree:
    - 10 → root
    - 20 → right of 10
    - 30 → right of 20 → imbalance at 10 (BF = -2) → **RR Rotation**
    - Tree after rotation: 20 (root) → 10 (left), 30 (right)

Example – Deletion:

- Delete 10 from above tree:
    - 20 becomes root → 30 remains right child → tree remains balanced (BF in [-1, 0, +1])

Impact on Performance:

- **Height maintained as O(log n)**
- Search, insertion, deletion → **O(log n)**

---

2. 2-3 Tree:

- Each node can have:

    - **2-node:** 1 key, 2 children
    - **3-node:** 2 keys, 3 children
- All **leaves are at the same level**, maintaining balance automatically.
- **Insertion:** Add key to leaf → if overflow occurs (more than 2 keys), **split node** and propagate middle key up.
- **Deletion:** Remove key from leaf → if underflow occurs (less than 1 key), **merge or redistribute** keys from sibling.

Example – Insertion:

- Insert keys: 10, 20, 30
    - 10 → root
    - 20 → 2-node becomes 2-key node (10,20)
    - 30 → overflow → split → 20 becomes root, 10 left child, 30 right child

Example – Deletion:

- Delete 10 → 2-node with single key 30 remains → tree remains balanced

Impact on Performance:

- Maintains height **O(log n)** automatically
- Search, insertion, deletion → **O(log n)**

Comparison of Balance Mechanisms:

Feature	AVL Tree	2-3 Tree
Balance Mechanism	Balance factor, rotations	Node splitting and merging

Feature	AVL Tree	2-3 Tree
Node Keys	1 key per node	1 or 2 keys per node
Height	Strictly balanced	All leaves at same depth
Operations Complexity	O(log n)	O(log n)
Implementation	More complex (rotations)	Simpler to maintain height